THESIS FOR THE DEGREE OF LICENTIATE OF PHILOSOPHY

Content Security for Web Applications

DANIEL HAUSKNECHT

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY AND GÖTEBORG UNIVERSITY

Göteborg, Sweden 2016

Content Security for Web Applications
DANIEL HAUSKNECHT

© 2016 Daniel Hausknecht

Technical Report 147L
ISSN 1652-876X
Department of Computer Science and Engineering
Research group: Language-based security

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY and GÖTEBORG UNIVERSITY
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Printed at Chalmers
Göteborg, Sweden 2016

Abstract

This thesis puts the focus on security problems related to web applications and web browsers by analyzing real-world web applications and modern client-side security mechanisms. For the latter, we mostly look at practical issues related to Content Security Policy (CSP) enforcement in web browsers.

First, we inspect password meters and password generators implementations on the web in a large scale empirical study. After discussing current practices and security concerns, we develop a generic framework for integrating password meters and generators in a secure way. We implement this framework solely based on today's existing browser technologies and demonstrate its effectiveness with a real world password meter.

Browsers come with frameworks to add functionality through browser extensions. By design, extensions are very powerful and can access and modify every part of visited web pages, from HTTP headers to a page's DOM. This also means security measures can be weakened or even removed completely. We investigate if and how browser extensions abuse their power by analyzing a large set of real-world browser extensions. We implement a mechanism which allows web servers to react to CSP header modifications by browser extensions.

Last, we shed light on CSP in the context of data exfiltration and the dispute in the security community whether CSP is meant to protect from it. We analyze the practical implications through an empirical study on DNS and resource prefetching mechanisms in web browsers allowing data exfiltration in the face of CSP. Finally, we discuss different possible research directions to limit data exfiltration attacks in the future.

Acknowledgments

This thesis is a product of many events, hard work and great times.

First, I want to thank Daniel Schreckling, Bastian Braun and the rest of the Passau crew who introduced me science and thaught me the very basics of scientific working. I am very grateful that you introduced me to Andrei and helped me to get my current position.

I want to thank Andrei Sabelfeld as a great supervisor. You manage to combine focused hard working with humor and a generally relaxed atmosphere, just to mention a few things. You learned me a lot and I am very much looking forward to the next coming years to be productive and to have even more fun together.

A special thanks goes also to Steven Van Acker and Jonas Magazinius who I have/had the honor to work together with. Thanks for our great discussions and productive times. I enjoyed our collaborations and am looking forward to do more in future.

Thanks to all my office mates but also the rest of the corridor and everyone else from Chalmers I like to hang out with. You are not just colleagues but also my friends who made me love living in Sweden from the first day on.

Mein Dank geht auch an meine Eltern und Brüder, die mich (meist aus der Ferne) in allem unterstützen, die ich manchmal ein bißchen vermisse aber auf die ich mich jedesmal umso mehr freue wieder zu sehn.

Last, I want to thank the love of my life Eszter. Thank you for your patience and support, all the adventures we had together and those which are yet to come, but most of all for you loving me! Nagyon szeretlek téged, Eszterem!

CONTENTS

CHAPTER

ONE

INTRODUCTION

The Internet, in particular the World Wide Web (WWW or just "the web"), has become an integral part of our daily lives. We use the web to search for information, watch online videos or to connect to friends in social networks. We even manage our finances online: we book a hotel room giving away our credit card number and transfer money through our bank's web interface. Short, the web provides a wide range of useful services with fundamentally different purposes. The big advantage of web services: users can access them from anywhere at any time. All that is needed is a web browser and an Internet connection.

Also application developers are motivated to use the web and pushing their services online. Web applications are platform independent, i.e. their application will execute in a Chrome browser on a Windows machine the same way as when using a Firefox on Linux or an Android powered device. Web applications can be updated instantly. All a developer needs to do is to update the code on the web server. When accessing the web application, web browsers automatically request the update version. One of the biggest advantages is probably the ease to include third-party content, a feature heavily used in practice [5]. A web developer does not need to "re-invent the wheel" but can just use resources provided by others. Some of these resources are less directly visible to users than others. An embedded YouTube video is easy to identify, whereas when a developer includes JavaScript program code to implement a web application, users are unlikely to realize that the code is served by a third party.

As there are numerous advantages of the web, there are many challenges, too. Let us look at a seemingly simple example, a registration page for a web service. Besides data like the user's name, commonly also a password is required. The password serves later as the ultimate secret which protects access to the account and naturally, it is advisable to choose a strong password. Some services support their users with choos-

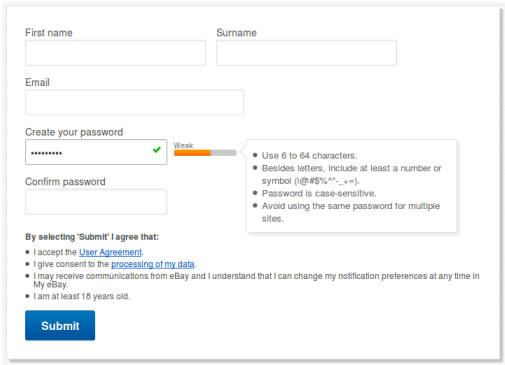

Fig. 1: Password meter on registration page for *ebay.co.uk*

ing a strong password by including so called *password meters* on their registration page. A password meter is a tool to measure the strength of a password, i.e. if it is too short, too easy to guess or actually good enough, the password meter presents this feedback to the user. A typical instance of such a registration page is the one by *ebay.co.uk* which we show in Figure 1.

But a strong password is not the only challenge here. There are many other especially security related issues which must be addressed: Where does the code for the password meter come from? Was it written by the service providers or included from a third party? Does it only assess the password strength within the browser or does it also send the password to a server? Is it the service provider's server or the one of a third party? Are other scripts used on the same web page? Do these scripts read out the password and if, what do they do with it? On submission, are the user credentials transmitted to the intended server and how? Is a secure connection used? How is the password stored on the server side? Is it stored in clear text, readable for everyone, or is it in some way obfuscated?

Many more questions could be asked. But to find answers, we need to learn how security can be put at risk, what is possible and how it can be done. We therefore switch to a different perspective: the view of a bad guy.

1 Attacking web applications

The "bad guys" have different names like hackers, attackers, adversaries and many more. Their intentions on the web are manyfold. Some want to directly make money, other want to steal data which can be sold on black markets, again others just want to destroy because they know how

to do it. The methods to achieve the goals can vary tremendously based on whatever the intentions are. Though we won't be able to cover all of them here, several techniques are more common and better established than others. The Open Web Application Security Project (OWASP) [7] maintains a list of the top ten most common attacks against web applications [8] and gives guidance how to counter them.

1.1 SQL injection attacks

Some attacks turn on web servers, for example, by trying to find security holes in the server's system implementation or configuration in hope to gain access to the server. Web administrators therefore frequently update their system software and adjust server configurations. Another famous attack which targets flaws in web application programming itself is the so called *SQL injection attack*.

To explain this kind of attack, let us assume there is a web page with the URL `http://example.com/profile.php` which displays a user's profile information. The essential part of the server-side code is shown in Listing 1.1.

```
1 $name = $_GET["name"]
2 $SQL_query = "SELECT * FROM Users WHERE name = "+$name;
3 echo getProfileFromDB($SQL_query);
```

Listing 1.1: SQL injection vulnerable web application

When the web page is requested, through for example

`http://example.com/profile.php?name=Daniel`,

the user's name is read from the URL parameter (line 1). With this name, a new SQL query statement is created (line 2) which is used to retrieve the profile information from the database (line 3).

Note that the value of the URL parameter can be basically freely chosen. This means, an attacker can decide to send any name in the web page request. In fact, an attacker is not even bound to send a legitimate name but can also send a string which effectively manipulates the database query. For example, an attacker can send the request

`http://example.com/profile.php?name=anything OR 1=1`.

This will result in a SQL query string

`SELECT * FROM Users WHERE Name = anything OR 1=1;`

which means either the profile name is "`anything`" or 1 equals 1. A tautology which by definition holds for every profile in the database. Consequently, all profiles are retrieved and displayed to the attacker in

the resulting web page. An attacker is of course not limited to only reading from a database but can also delete single entries or, even worse, empty the whole database.

1.2 Cross-site scripting (XSS) attacks

Other attacks target the client side and try, e.g., to steal login credentials from particular users or to steal otherwise sensitive user information. One of the most prevailing type of such attacks is *cross-site scripting* (XSS).

OWASP defines XSS attacks as "a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites" [6]. An important characteristic is that an attack does not leverage security holes in browsers nor in on server installed software, but bugs in scripts used by a web application.

Let us consider a web page with the URL `http://example.com/hello.php` which HTML code is generated using the PHP code as shown in Listing 1.2. When the web page is requested with

`http://example.com/hello.php?name=Daniel`,

the value from the URL parameter `name` is injected into the resulting HTML code. In our case, the string "Hello Daniel!" is generated.

```
1 <?php echo "Hello " . $_GET["name"] . "!"; ?>
```

Listing 1.2: XSS vulnerable web page

As in the SQL injection attack example before, the value of the URL parameter can be basically freely chosen. This means, an adversary can decide to send JavaScript code in the web page request, instead of a legitimate name. A possible such forged request is

```
http://example.com/hello.php?
    name=<script src="http://evil.com/attack.js"></
        script>
```

in which the `name` parameter in the URL request is a HTML `<script>` element which points to a JavaScript file hosted on the adversary's web server. The generated HTML code is shown in Listing 1.3.

```
1 Hello <script src="http://evil.com/attack.js"></script>!
```

Listing 1.3: HTML code generated by XSS vulnerable PHP script in Listing 1.2

The server-side PHP code does not sanitize the input originating from the client. As a result, an attacker is able to inject any code into the web

page by crafting URLs as just shown. To launch an actual attack, the adversary only needs to distribute the malicious URL as a web link on, e.g., a forum or in an email and wait until a victim clicks on it.

There are many other ways to start XSS attacks. In general, there are two different XSS types [9], *non-persistent (reflected) and persistent (stored) XSS*, each of which can be either server- or client-side based.

Non-persistent (reflected) XSS attacks In a non-persistent XSS attack, the malicious script is injected based on dynamic data, e.g. the URL of a web page request.

In our example above, the parameter value is taken directly from the request by the server-side script and the attack is only present for forged URLs while the web application behaves normally otherwise. Because the attack goes from the attacker to the server to finally run in a victims browser, the attack is reflected on the web server and non-persistent XSS attacks are therefore also called reflected XSS attacks.

In the example, the code injection happens on the server side. The same attack can also be performed on the client side only, e.g., when JavaScript code processes the `document.location` property from the Document Object Model (DOM) to for example retrieve the user name from the URL parameter. The access of the DOM led to the name DOM-based XSS for non-persistent client-side XSS attacks.

Persistent (stored) XSS attacks In a persistent XSS attack, the malicious script is injected using permanently stored data. Therefore, persistent XSS attacks are also called stored XSS attacks.

Usually, users do not have the privileges to directly access a web server's file system and to store data. But as it is for example the purpose of forums, users can write entries which are then permanently stored in the web application's database. Every time a user visits the forum, previously written entries are retrieved from the database and injected into the web page. An adversary can write a forum entry similar to as in Listing 1.3. If the forum server does not sanitize the entry properly, the HTML `<script>` element is included into the forum page persistently. Every time the page is visited, the script `attack.js` is loaded and the XSS attack is performed.

As it is for reflected XSS attacks, malicious code can also be injected only on the client side. The attacker's code can be stored in the client, for example, through the local storage API in web browsers. One possible scenario is that the attacker is able to replace benign content in the local storage with malicious code. Every time the locally stored data is loaded to update a web page, the malicious code is injected and the XSS attack is performed.

At this point it is worth mentioning that malicious code can be injected in many other ways such as intercepting and modifying network

traffic [1]. However, those techniques are not subject to web application security but, as in this case, network security and we therefore do not discuss them further in this thesis.

2 Protecting web applications

The general problem is that most end users learned how to browse the web but do not necessarily understand the technologies themselves and the risks coming with them. End users are likely to not identify potential security risks and to discover when they are under attack. Therefore, it is the responsibility of web developers to provide as much protection to end users as possible. The challenge for web service providers is that, though they do control their own servers, they have basically no control over the client's browser environment. It is unpredictable whether the service is accessed on a private or a public computer, which web browser is used and which version is installed, if the service is embedded into the context of another web service and whether this service can be trusted.

2.1 Prepared statements

Despite the lack of client-side control, web service administrators are not completely powerless. They can implement verification techniques on their servers to check for validity and legitimacy of service usage. For example, SQL injections can be easily prevented through using a programming technique called *prepared statements* (e.g. [2] for PHP). The SQL injection vulnerable PHP code from our previous example in Listing 1.1 can be secured through code similar to as shown in Listing 1.4. First, the SQL statement is prepared with a placeholder for the user's profile name represented through a '`?`'. In line 3, the placeholder is replaced by the actual name as received in the request URL parameter. In contrast to the vulnerable code, the whole input is now interpreted as the user name. A possible SQL injection attempt as before is now interpreted as querying for a profile with user name "`Daniel OR 1=1`". Thus, an attacker can no longer influence the SQL statement.

```
1 $stmt =
2    $dbh->prepare("SELECT * FROM Users where name = ?");
3 $stmt->execute(array($_GET['name']));
4 echo $stmt->fetch();
```

Listing 1.4: PHP code using prepared statements

Note that an attacker is still able to choose the user name freely. SQL statements cannot prevent this. To protect from this, some kind of access control mechanism is needed, e.g. through logging in to the web service.

Additionally to server-side protections, web developers and browser vendors started to closely work together and to integrate security measures into web browsers. With all major browser vendors implementing certain security standards, end users profit from this development without the need to understand the technology. Web developers, on the other hand, can rely on browsers to enforce these mechanisms.

2.2 Same-origin Policy (SOP)

One significant security feature implemented in all browsers is the *Same-Origin Policy* (SOP). The basic idea is to allow connections for sending and receiving data only to servers with the same origin as the currently loaded web page. Usually, origins are defined to be the same if the used protocol, the domain name and the port match [4] (note that there exists no official standard). In case of a XSS attack when an attacker tries to leak sensitive data from for example a web page hosted on `http://example.com` to a server `http://attacker.com`, the connection is blocked because the origins differ. In it is pure form however, SOP is too restrictive. It would neither allow to embed a YouTube video on a web page nor to use scripts as, e.g., provided by Google Analytics to get page usage statistics. All of those cases are common scenarios. Therefore, the SOP is in practice relaxed to allow loading various types of third party resources, e.g. scripts and images from different origins. In case of Google Analytics, the relaxation allows to firstly fetching the JavaScript code to collect user statistics. Opening a direct connection to Google servers and sending the data would still be prohibited by the SOP. Therefore, the common trick to load an image of simply one pixel is used. Fetching images is not blocked by the SOP and the user statistics can be send with the request as URL parameters.

2.3 Content Security Policy (CSP)

In case of analytics scripts, the exception of the SOP to send an image request to a third party is wanted by web application administrators and accepted by the end users. However, adversaries are naturally also able to use those SOP exceptions and it can be used to leak sensitive data as part of an XSS attacks.

To demonstrate the issue, let us look at the code of an exemplary registration page. Besides the registration form, the page also includes a password meter which shows different images indicating if a chosen password is weak or strong. Let us assume an attacker can somehow inject JavaScript code into this registration page. The page's full HTML code is shown in Listing 1.5 with the attacker-injected code from line 8 - 19.

```
<form id="register_form" method="POST" action="welcome.
    php">
  <h1>Register</h1>
  <input type="text" id="user_name" name="user_name"/>
  <input type="password" id="password" name="password"
      onkeyup="check()"/>
  <input type="submit" value="Register"/>
</form>

<script>
function leakData() {
  var user_name = document.getElementById("user_name").
      value;
  var password = document.getElementById("password").
      value;

  var img = document.createElement("IMG");
  img.src = "https://evil.com/"
              +"?username="+user_name+"&password="+
                  password;
}

document.getElementById("register_form").onsubmit =
    leakData;
</script>

<script src="https://friendly.com/password_meter.js"></
    script>
<script>
function check() {
  var password = document.getElementById("password").
      value;
  var result = assessPassword(password);

  var img = document.getElementById("img");
  if (result === "strong") {
    img.src = "https://example.com/strong.png";
  } else {
    img.src = "https://example.com/weak.png";
  }
}
</script>
<img id="img" />
```

Listing 1.5: XSS attack execution on a registration page

The web page's legitimate registration form (line 1 - 6) includes a text input field for a user name and a password as well as a submit button to send the account data to the web server. The attacker's code defines a new JavaScript function `leakData` which first reads out the user name and the password from the registration form (lines 10 - 11). Next, the script uses the previously described trick to circumvent the SOP as follows: In line 13, a new image DOM object is created. The two following lines set the image source URL. Most importantly in line 15, the user name and password are attached to the image URL as parameters. Assigning the new source also triggers loading the image and the browser sends the respective request to `evil.com` including the just attached parameters. The function is set to be executed when the register button is used (line 18), effectively leaking the account data to the attacker-controlled server.

As the code in Listing 1.5 demonstrates, SOP is not sufficient to prevent XSS attacks in practice. Therefore, the World Wide Web Consortium (W3C) [13] started to develop a more fine grained security mechanism which allows to distinguish between trusted and untrusted web sources: *the Content Security Policy* (CSP) [12].

CSP deployment The basic idea of CSP is to whitelist all trusted sources from which content is loaded into a web page. These sources are categorized by their type of content they provide in so called *directives*. Example directives are `script-src` for script sources or `img-src` for image sources. Notable is also the `default-src` directive which is applied in case a specific directive is not defined in a policy. Inline scripts and `eval` functions are disabled by default but can be re-enabled through `'unsafe-inline'` and `'unsafe-eval'`, respectively.

CSP policies are defined on the server side and sent either as a HTTP response header or alternatively as a HTML `<meta>` tag with `http-equiv` attribute. Web browsers implementing the CSP standard enforce the policy.

Let us re-visit the registration page example in Listing 1.5. The goal is to define a CSP policy for this legitimate part of the web page. To have the most effective policy, it is desirable to be as restrictive as possible. Therefore, we want to allow by default no resources to be loaded from any source. This is achieved by setting the value of the `default-src` directive to `'none'`. Besides the registration form, there is a second and non-attacker part which implements the password meter feature (lines 21 - 35). First, an external script from `friendly.com` is included which implements the basic password measuring functionality. The domain and the script are trusted and added to the CSP with `script-src https://friendly.com`. This second part of the page also contains a legitimate inline script in lines 22 - 34. We therefore needs to be re-enable inline scripting. Last, we want to show different images from `example.com` to illustrate the strength of a chosen password. The domain `example.com`

is whitelisted in the `img-src` directive. The complete resulting CSP is shown in Listing 1.6.

```
1 default-src 'none'; script-src https://friendly.com '
    unsafe-inline'; img-src https://example.com;
```

Listing 1.6: CSP policy for web page in Listing 1.5

When applying the CSP in Listing 1.6, we can observe that the policy is permissive enough to load all legitimate sources, i.e. the password meter script and the images, as intended. But there is one drawback to the CSP policy in Listing 1.6: the CSP does not only allow the execution of the legitimate inline script in lines 22 - 22 but also the attacker's script in lines 8 - 19. At this point, we need to remember that the attacker tries to leak data by requesting a image from `http://evil.com`. But since the CSP policy restricts image source to only `http://example.com`, i.e. `http://evil.com` is not whitelisted, the actual image request is blocked by browsers. Consequently, even though the attacker's script is allowed to run, the overall attack is effectively prevented by the CSP policy.

Still, web attacks are generally not impossible. The reason is that existing security technologies are either not sufficient even when applied correctly or do have practical shortcomings. We have seen an example with SOP already. But also other mechanisms like authenticated and encrypted communication does not always help. Note that in the previous example only HTTPS is used as the protocol, even by the attacker. However, this just means that the connection to the attacker's server is secured, but it does not prevent data from being leaked. Last but not least, CSP's adoption is relatively slow because it is quite hard to come up with effective CSP policies for more complex web pages [14]. For example, its static definition can quickly lead to conflicts with the fundamental dynamic character of the web. Obviously, web security research is not at the end of its road and more needs to be done to protect services and end users, and to make existing mechanisms more practical.

3 Thesis overview

We will now highlight three unattended issues in web security, derive questions which motivate our research, and summarize how this thesis contributes to answer them.

3.1 Online password meters and password generators

Motivation As we discussed earlier, most web services control user access through passwords. Naturally, choosing a good password is the A and O for a user to protect the own account. Is a password too simple, it

is easy for attackers to guess; is it too complex, one can hardly remember it for the next login.

Fortunately, online password meters and password generators emerged. A password meter is a tool which allows to measure the strength of passwords and gives feedback to users whether a selected password is good enough or to weak. Web services integrate password meters on registration pages to support the selection of strong passwords. To make the step of coming up with a strong password in the first place easier, a password generator is a tool that creates new passwords for users. Both tools can be found online as web services. Though they are convenient to use, passwords play a key role in web security and usage of those tools should not put online accounts at risk. In this regard, we can formulate the following two research questions:

Research question How are password meters and password generators implemented on the web? How can web developers integrate them in a safe and secure way?

Thesis contribution We conducted a large-scale empirical study to analyze password meters and password generators on the web. For this, we automatically crawled the web in search for password meters and password generators as either stand-alone services or as part of online registration pages. The results are analyzed for security relevant properties such as third party code inclusion, password transmission over the network and whether this transmission was in clear text. Based on our findings, we specify desired properties for a safe and secure execution environment of online password meters and generators. As a proof of concept, we implement SandPass and demonstrate its effectiveness using the password meter provided by the Swedish Post and Telecommunication Agency (now hosted by "Myndigheten för samhällsskydd och beredskap").

Statement of contributions This paper was co-authored with Steven Van Acker and Andrei Sabelfeld. Daniel was mainly responsible for developing the framework SandPass and writing the respective sections.

The respective chapter was published as a paper in the proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY) 2015.

3.2 CSP modifications through browser extensions

Motivation Browser extensions, sometimes also called add-ons, are a convenient way to add functionality to web browsers. They have gained wide popularity, some counting millions of users. To fulfill their tasks, they often need the full capabilities of browsers, for example injecting content into loaded web pages or even modifying web requests and responses. Extensions can inject content directly into a web page. If the

source of the content is however blocked by a CSP, extensions can relax the web page's CSP to also whitelist the source in question. A CSP is defined by web service providers with the best intentions to protect their users and to exclude certain sources from the CSP on purpose. With browser extensions being able to modify a CSP, the security of a web application can be weakened but without the consensus of service providers.

Research question Do browser extensions make active use of their capability to modify CSPs? How do browsers enforce a web page's CSP on extension injected resources? Is there a way for browsers to support extensions modifying CSP to work properly, but at the same time to allow web services to react to the affected security on provided web pages?

Thesis contribution We automatically downloaded over 25853 Chrome browser extensions. In the paper, we analyze them for behavioral patterns (e.g. modification of HTTP headers) and categorize them into three different basic vulnerability classes for affected web pages: *third party code inclusion*, *enabling of XSS* and *user profiling*. We also analyze web browsers how they handle resources injected into web pages by extensions with respect to CSP. Except for Firefox, extension injected resources are not restricted by a web pages CSP. We develop a mechanism that allows web service providers to react to CSP modifications made by browser extensions. Providers can either endorse the change or to reject in case the they see the service's security at risk. We conduct a case study based on Gmail and the browser extension Rapportive to demonstrate the effectiveness of our prototype implementation.

Statement of contributions This paper was co-authored with Jonas Magazinius and Andrei Sabelfeld. Daniel was mainly responsible for the manual analysis of Chrome browser extensions, the implementation of the CSP endorsement mechanism for Firefox and Chrome browsers, and for writing the respective sections.

The respective chapter was published as a paper in the proceedings of the 12th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA) 2015.

3.3 Data exfiltration in the context of CSP

Motivation Among researchers and web developers there exists an ongoing dispute whether CSP is meant to prevent data exfiltration. While some say CSP is designed to control resource injections only, others point to features such as the `form-data` directive and argue for CSP to limit data exfiltration. Unfortunately, the standard itself is rather vague on this point.

Additionally, certain browser features such as for performance seem not to be covered by CSP at all. For browsers to perform faster, browser vendors came up with different techniques to resolve domain names in advance or even prefetch page content before a web page is actually requested. With every domain name resolution or resource prefetching, a respective request is sent automatically by browsers without any human interaction. In fact, prefetching requests are ordinary web requests and, if done properly, can be used to create a special communication channel between browser and server (in contrast to between web page and server).

Research question Which are the different viewpoints on the very purpose of CSP in the security community? Can adversaries exploit DNS resolution and resource prefetching to leak data from browsers? Can these communication channels be restricted through CSP?

Thesis contribution We report on the discord of researchers and web developers if CSP is meant to mitigate data exfiltration attacks. After providing the necessary background on CSP, domain name service (DNS) and prefetching techniques, we conduct a systematic case study on DNS and resource prefetching in various browser implementations. We demonstrate that it is under certain conditions possible to exploit browser performance features to exfiltrate data in the face of CSP. We conclude by discussing several possible research directions to mitigate the threat of data exfiltration attacks in the future.

Statement of contributions This paper was co-authored with Steven Van Acker and Andrei Sabelfeld. Daniel was mainly responsible for writing the sections explaining the required technical background, the attacker model, the discussion of possible measures and related work.

The respective chapter will be published as a paper in the proceedings of the 11th ACM Asia Conference on Computer and Communications Security (ASIACCS) 2016.

References

1. Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell, and Dawn Song. Towards a Formal Foundation of Web Security. In *Proceedings of CSF 2010*, 2010.
2. The PHP group. PHP: Prepared statements and stored procedures - Manual. `http://php.net/manual/en/pdo.prepared-statements.php`. last visited: 2015-08-12.
3. LastPass. The LastPass Blog. `https://blog.lastpass.com/2015/06/lastpass-security-notice.html/`. last visited: 2015-08-12.
4. Mozilla. Same-origin policy - Web security — MDN. `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`. last visited: 2015-08-12.

5. Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.
6. OWASP. Cross-site Scripting (XSS). `https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29`. last visited: 2015-08-12.
7. OWASP. OWASP. `https://www.owasp.org/index.php/Main_Page`. last visited: 2015-08-12.
8. OWASP. OWASP Top Ten Project. `https://www.owasp.org/index.php/Top10#OWASP_Top_10_for_2013`. last visited: 2015-08-12.
9. OWASP. Types of Cross-Site Scripting. `https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting`. last visited: 2015-08-12.
10. Sophos. Cheating site Ashley Madison breached by hackers threatening to expose users. `https://nakedsecurity.sophos.com/2015/07/20/cheating-site-ashley-madison-breached-by-hackers-threatening-to-expose-users/`. last visited: 2015-08-12.
11. Sophos. Domino's Pizza hacked, customer database held to ransom. `https://nakedsecurity.sophos.com/2014/06/16/dominos-pizza-hacked-customer-database-held-to-ransom/`. last visited: 2015-08-12.
12. W3C. Content security policy 2.0. `http://www.w3.org/CSP`. last visited: 2015-08-12.
13. W3C. World Wide Web Consortium (W3C). `http://www.w3.org`. last visited: 2015-08-12.
14. M. Weissbacher, T. Lauinger, and W. Robertson. Why Is CSP Failing? Trends and Challenges in CSP Adoption. In *RAID*, 2014.